МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Линейные структуры данных: стек, очередь, дек

Вариант 9-д

Студент гр. 8304 _____ Масалыкин Д.Р.
Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2019

Цель работы.

Познакомиться с часто используемыми на практике линейными структурами данных, обеспечивающими доступ к элементам последовательности только через её начало и конец, и способами реализации этих структур, освоить на практике использование стека, очереди и дека для решения задач.

Постановка задачи.

В заданном текстовом файле F записан текст, сбалансированный по круглым скобкам:
< текст > ::= < пусто > | < элемент > < текст >
< элемент > ::= < символ > | (< текст >)
где < символ > – любой символ, кроме (,). Для каждой пары соответствующих открывающей и закрывающей скобок вывести номера их позиций в тексте, упорядочив пары в порядке возрастания номеров позиций:
а) закрывающих скобок; б) открывающих скобок.

Описание алгоритма.

Программа считывает данные и отправляет их на проверку: в стек заносится форма скобочной конструкции, затем символы считываются по 1 и сравниваются с тем, что находится на вершине стека. Если они равны – из стека извлекается символ, если же нет, то ищется правило с данными символами. При его существовании символ на вершине стека заменяется на это правило, а при отсутствии – возвращается номер текущего символа, как адрес ошибки. Всё повторяется до тех пор, пока не будут считаны все символы и стек не опустеет.

Спецификация программы.

Программа предназначена для валидации скобочной конструкции. Программа написана на языке C++ с использованием CLI. Входными данными являются строки из файла, который подается как аргумент командной строки.

Тестирование.

Таблица 1 – Результаты тестирования программы

Input	Output
()	1 2
ASD (asdasd)	5 12
ASD (asdas((d)))	5 16 |11 15 |12 14
ASD (asdas((d))) (text(and text))	12 14|11 15|5 16|23 32|18 33|

ASD (asdas((d))) (text(and text))	5 16|11 15|12 14|18 33|23 32|

Анализ алгоритма.

Алгоритм работает за линейное время от размера строки.

Описание функций и СД.

Класс **DynamicStack** реализует структуру стека, а также методы для работы с ним.

Стандартные методы для работы со стеком:

```
T pop();
void void push(const T& data);
bool isEmpty();
T onTop();
```

Выводы.

В ходе работы были приобретены навыки работы со стеком, изучены методы работы с ним (объявлять, заносить в него переменных и забирать их).

Приложение А. Исходный код программы.

dynamicstack.h

```
#ifndef DYNAMICSTACK_H
#define DYNAMICSTACK_H

#include<memory>
#include <iostream>
#include"node.h"

template <typename T>
class DynamicStack
{
public:
    typedef std::shared_ptr<DynamicStack> DynamicStackP;

private:
    typename Node<T>::NodeP head_ = nullptr;
    typename Node<T>::NodeP tail_ = nullptr;
    int elementsCount_ = 0;

public:
    DynamicStack() = default;
    DynamicStack(std::initializer_list<T> init);

    void push(const T& data);
    void push(const DynamicStack& stack);

    void pushBack(const T& data);
    void pushBack(const DynamicStack& stack);

    T pop();
    T onTop();

    size_t size();
    bool isEmpty();
    void clear();
};

template <typename T>
DynamicStack<T>::DynamicStack(std::initializer_list<T> init)
{
    for(auto value = init.begin(); value != init.end(); value++)
    {
        pushBack(*value);
    }
}

template <typename T>
void DynamicStack<T>::push(const T& data)
{
    typename Node<T>::NodeP newNode(new Node<T>);
    newNode->setData(data);

    if(head_ == nullptr)
    {
        head_ = newNode;
        tail_ = newNode;
    }
    else
    {
        newNode->setNext(head_);
        head_->setPrev(newNode);
        head_ = newNode;
    }

    elementsCount_ += 1;
}

template <typename T>
void DynamicStack<T>::push(const DynamicStack& stack)
{
    typename Node<T>::NodeP buf = stack.tail_;
```

```
    while(buf != nullptr)
    {
        T dataBuf = buf-

        >data(); push(dataBuf);

        buf = buf->prev();
    }
}

template <typename T>
void DynamicStack<T>::pushBack(const T& data)
{
    typename Node<T>::NodeP newNode(new Node<T>);
    newNode->setData(data);

    if(head_ == nullptr)
    {
        head_ = newNode;
        tail_ = newNode;
    }
    else
    {
        newNode->setPrev(tail_);
        tail_->setNext(newNode);
        tail_ = newNode;
    }

    elementsCount_ += 1;
}

template <typename T>
void DynamicStack<T>::pushBack(const DynamicStack& stack)
{
    typename Node<T>::NodeP buf = stack.head_;
    while(buf != nullptr)
    {
        T dataBuf = buf->data();

        pushBack(dataBuf);

        buf = buf->next();
    }
}

template <typename T>
T DynamicStack<T>::pop()
{

    typename Node<T>::NodeP buf  = head_;

    head_ = buf->next();

    buf->setNext(nullptr);

    elementsCount_ -= 1;

    if(head_ == nullptr)
    {
        tail_ = nullptr;
    }

    return buf->data();
}

template <typename T>
T DynamicStack<T>::onTop()
{
    return head_->data();
}

template <typename T>
bool DynamicStack<T>::isEmpty()
{
    return (elementsCount_ == 0);
}

template <typename T>
void DynamicStack<T>::clear()
```

```
{
    head_ = nullptr;
    tail_ = nullptr;
    elementsCount_ = 0;
}

template <typename T>
size_t DynamicStack<T>::size()
{
    return elementsCount_;
}

#endif // DYNAMICSTACK_H
```

node.h

```
#ifndef NODE_H
#define NODE_H

#include<memory>

template <typename T>
class Node
{
public:
    typedef std::shared_ptr<Node> NodeP;
    typedef std::weak_ptr<Node> NodeWP;
private:
    T data_;
    NodeWP prev_;
    NodeP next_ = nullptr;
public:
    Node() = default;

    void setData(T data);
    T data();

    void setPrev(NodeP node);
    NodeP prev();

    void setNext(NodeP node);
    NodeP next();
};

template<typename T>
void Node<T>::setData(T data)
{
    data_ = data;
}

template<typename T>
T Node<T>::data()
{
    return data_;
}

template<typename T>
void Node<T>::setPrev(NodeP node)
{
    prev_ = node;
}

template<typename T>
typename Node<T>::NodeP Node<T>::prev()
{
    if(prev_.expired())
    {
        return nullptr;
    }
    else
    {
        return NodeP(prev_);
    }
}

template<typename T>
void Node<T>::setNext(NodeP node)
{
    next_ = node;
}

template<typename T>
typename Node<T>::NodeP Node<T>::next()
{
```

```
    return next_;
}

#endif // NODE_H
```

main.cpp

```
#include <iostream>
#include "DynamicStack.h"
#include <fstream>
#include <string>

void back(std::ifstream* file);
void front(std::ifstream* file);
void print_reverse_stacks(DynamicStack<int>* stack_open, DynamicStack<int>* stack_close);

int main(int argc, char* argv[]) {
    if(argc == 1){
        std::cout<<"No input file"<<std::endl;
        return 1;
    }
    std::ifstream file;
    file.open(argv[1]);
    if(!file.is_open()){
        std::cout<<"Can't open file"<<std::endl;
        return 1;
    }

    int flag;
    std::cout<<"back(1) or front(0)"<<std::endl;
    std::cin>>flag;
    if(flag)
        back(&file);
    else
        front(&file);
    file.close();
    return 0;
}

void back(std::ifstream* file){
    DynamicStack<int> stack;
    std::string inp_str;
    while(getline(*file, inp_str)){
    for(unsigned int i = 0; i < inp_str.size(); i++){
        if(inp_str[i] == '('){
            stack.push(i+1);
        }
        else if(inp_str[i] == ')'){
            std::cout<<stack.pop()<<" "<<i+1<<"|";
        }
    }
   }
}

void print_reverse_stacks(DynamicStack<int>* stack_open, DynamicStack<int>* stack_close){
     DynamicStack<int> tmp_open, tmp_close;
     while(stack_open->size() > 0){
             tmp_open.push(stack_open->pop());
     }
     while(tmp_open.size() > 0){
             std::cout<<tmp_open.pop()<<" "<<stack_close->pop()<<"|";
     }
}

void front(std::ifstream* file){
    DynamicStack<int> stack_open, stack_close;
    std::string inp_str;
    while(getline(*file, inp_str)){
    for(unsigned int i = 0; i < inp_str.size(); i++){
        if(inp_str[i] == '('){
            stack_open.push(i + 1);
        }
        else if(inp_str[i] == ')'){
         stack_close.push(i+1);
         }
     if(stack_open.size() == stack_close.size() && stack_open.size() > 0 &&
stack_close.size() > 0){
         print_reverse_stacks(&stack_open, &stack_close);
     }
    }
    }
}
```